TFE4171 - High-level Data Link Control Project

Jens Fredrik Bergem & Toushif Islam Shoueb

April 2025

Contents

List of Figures

List of Tables

1 Introduction

High-level Data Link Control (HDLC) is a data communication protocol, which supports frame error detection through a Cycle Redundancy Check (CRC). It also allows for abortion of frames, and continuously detects for overflows. This report aims to verify the functional behavior of a given HDLC module, using both immediate and concurrent assertions.

2 Part A

Note: A more detailed description of the Rx_SC register can be viewed in part B under section 3.2.1.

2.1 Show VerifyNormalReceive task. Explain which flags you verified, and how you checked the RX data buffer

Preconditions The conditions in which there is a normal receive is when the input size to the buffer is within 128 bytes, where 2 of the bytes are reserved for generated FCS bytes. Additionally, under a normal receive, the Rx_FCSen bit in the Rx_SC register is also enabled, and the flagsequence (01111110) should be generated.

The immediate assertions for normal receive can be viewed in figure 1. For both the Rx buffer and the Rx_SC, we read the registers through the *ReadAddress* task, and store the read data into the register ReadData. The Rx_SC is checked by doing bit comparisons between each read bit and the expected value as depicted in table 3. For the Rx buffer we loop through each byte, comparing the written data (data[i]) against the read data in the register. How many bytes to check is decided by the integer Size which represents the amount of bytes generated as input stimulus to the buffer.

```
70  task VerifyNormalReceive(logic [127:0][7:0] data, int Size);
71    logic [7:0] ReadData;
72    automatic int RXSC_error = 0;
73    automatic int RXbuff_error = 0;
74    wait(uin_hdlc.Rx_Ready);
75
76    // VERIFY DATA IN RXSC REGISTER
77    ReadAddress(RxSC, ReadData);
78    assert (ReadData [0]) else begin $display("VerifyNormalReceive     FAIL : Rx_Ready bit is not high (Immediate assertion)"); RXSC_error++; end
79    assert (!ReadData[1]) else begin $display("VerifyNormalReceive      FAIL : Rx_Drop bit is not low (Immediate assertion)"); RXSC_error++; end
80    assert (!ReadData [2]) else begin $display("VerifyNormalReceive     FAIL : Rx_FrameError bit is not low (Immediate assertion)"); RXSC_error++; end
81    assert (!ReadData [3]) else begin $display("VerifyNormalReceive     FAIL : Rx_AbortSignal bit is not low (Immediate assertion)"); RXSC_error++; end
82    assert (!ReadData [4]) else begin $display("VerifyNormalReceive     FAIL : Rx_Overflow bit is not low (Immediate assertion)"); RXSC_error++; end
83
84    // VERIFY DATA IN RX BUFFER
85    for (int i = 0; i < Size; i++) begin
86      ReadAddress(RxBuff, ReadData);
87      assert (ReadData == data[i]) else begin $display("VerifyNormalReceive
88      FAIL : Rx data buffer does not contain the correct value (Immediate assertion)"); RXbuff_error++; end
89    end
90
91    if (!RXSC_error && !RXbuff_error)
92      $display("VerifyNormalReceive     PASS : Rx databuffer and Rx_SC  contains the correct values (Immediate assertion)");
93    else
94      TbErrorCnt += (RXSC_error + RXbuff_error);
95  endtask
```

Figure 1: VerifyNormalReceive task (Immediate Assertion)

2.2 Show VerifyAbortReceive task. Explain which flags you verified, and how you checked the RX data buffer.

Preconditions The conditions in which there is an aborted receive are similar to a normal receive, except in this case there is an inserted abort-flag sequence.

In the case of an abort receive we expect the Rx buffer to be empty and that the Rx_Abort bit in the Rx_SC register is asserted. In this particular case, we also expect that the Rx_Ready bit is never asserted.

```
44  task VerifyAbortReceive(logic [127:0][7:0] data, int Size);
45    logic [7:0] ReadData;
46    automatic int RXSC_error = 0;
47    automatic int RXbuff_error = 0;
48
49    ReadAddress(RxSC, ReadData);
50    assert (!ReadData [0]) else begin $display("VerifyAbortReceive      FAIL : Rx_Ready bit is not low (Immediate assertion)"); RXSC_error++; end
51    assert (!ReadData[1]) else begin $display("VerifyAbortReceive       FAIL : Rx_Drop bit is not low (Immediate assertion)"); RXSC_error++; end
52    assert (!ReadData [2]) else begin $display("VerifyAbortReceive      FAIL : Rx_FrameError bit is not low (Immediate assertion)"); RXSC_error++; end
53    assert (ReadData [3]) else begin $display("VerifyAbortReceive      FAIL : Rx_AbortSignal bit is not high (Immediate assertion)"); RXSC_error++; end
54    assert (!ReadData [4]) else begin $display("VerifyAbortReceive      FAIL : Rx_Overflow bit is not low (Immediate assertion)"); RXSC_error++; end
55
56    for (int i = 0; i < Size; i ++) begin
57      ReadAddress(RxBuff, ReadData);
58      assert (ReadData == 0) else begin $display("VerifyAbortReceive FAIL : Rx data buffer is not zero (Immediate assertion)"); RXbuff_error++; end
59    end
60
61    if (!RXSC_error && !RXbuff_error)
62      $display("VerifyNormalReceive     PASS : Rx databuffer and Rx_SC contains the correct values (Immediate assertion)");
63    else
64      TbErrorCnt += (RXSC_error + RXbuff_error);
65  endtask
```

Figure 2: VerifyAbortReceive task (Immediate Assertion)

2.3 Show VerifyOverflowReceive task. Explain which flags you verified, and how you checked the RX data buffer.

Preconditions An overflow receive happens when more than 128 bytes are received.

In case of an overflow, assertions are made to check that the corresponding Rx_Overflow bit in the Rx_SC register is asserted. Additionally, we expect the value in the Rx buffer to correspond to the first 126 bytes send to the buffer.

```
96  task VerifyOverflowReceive(logic [127:0][7:0] data, int Size);
97    logic [7:0] ReadData;
98    automatic int RXSC_error = 0;
99    automatic int RXbuff_error = 0;
100   wait(uin_hdlc.Rx_Ready);
101
102   // VERIFY DATA IN RXSC REGISTER
103   ReadAddress(RxSC, ReadData);
104   assert (ReadData[0]) else begin $display("VerifyOverflowReceive   FAIL : Rx_Ready bit is not high (Immediate assertion)"); RXSC_error++; end
105   assert (!ReadData[1]) else begin $display("VerifyOverflowReceive   FAIL : Rx_Drop bit is not low (Immediate assertion)"); RXSC_error++; end
106   assert (!ReadData[2]) else begin $display("VerifyOverflowReceive   FAIL : Rx_FrameError bit is not low (Immediate assertion)"); RXSC_error++; end
107   assert (!ReadData[3]) else begin $display("VerifyOverflowReceive   FAIL : Rx_AbortSignal bit is not low (Immediate assertion)"); RXSC_error++; end
108   assert (ReadData[4]) else begin $display("VerifyOverflowReceive   FAIL : Rx_Overflow bit is not high (Immediate assertion)"); RXSC_error++; end
109
110   // VERIFY DATA IN RX BUFFER
111   for (int i = 0; i < Size; i++) begin
112     ReadAddress(RxBuff, ReadData);
113     assert (ReadData == data[i]) else begin $display("VerifyOverflowReceive FAIL : Rx data buffer does not contain the correct value (Immediate assertion)"); RXbuff_error++; end
114   end
115
116   if (!RXSC_error && !RXbuff_error)
117     $display("VerifyNormalReceive     PASS : Rx databuffer and Rx_SC contain the correct values (Immediate assertion)");
118   else
119     TbErrorCnt += (RXSC_error + RXbuff_error);
120 endtask
```

Figure 3: VerifyOverflowReceive task (Immediate Assertion)

2.4 Say whether the following statement is true or false: "assertions in the tasks mentioned above are all concurrent

False. All assertions above are immediate as they checked at current simulation times and not continously throughout the whole simulation.

2.5 Difference between immediate and concurrent assertions:

The main difference between immediate and concurrent assertions are that immediate assertions are evaluated at the current simulation time whereas concurrent assertions are evaluated over multiple units of time/clock cycles. In this report we have used both immediate and concurrent assertions, and for each specification we will explain the reasoning behind our decision to choose one over the other.

2.6 Show Rx flag sequence (behaviour of Rx FlagDetect). Explain it.

According to the specification, the flagsequence is defined as the bitstream *01111110*. On the receiving side we expect the Rx signal to follow this pattern at the start and end of a frame (assuming it's not an abort). This can be defined as a sequence given by !Rx ##1 Rx [*6] ##1 !Rx, where ##1 is a unit of delay and [*6] is equivalent to writing Rx ##1 Rx ##1 etc... When inserting the sequence into a property synchronized with a clock, ##1 is then defined as a delay of 1 clock cycle.

```
56    sequence Rx_flag;
57      !Rx ##1 Rx [*6] ##1 !Rx; //01111110
58    endsequence
```

Figure 4: Rx_flag sequence

2.7 Show Rx AbortSignal property. Explain it.

The abort-flag sequence is defined as *01111111*, and this sequence was defined as Rx_Abortdetect. Furthermore we used this sequence in the RX_AbortSignal property. When the sequence is detected, and RxValidFrame is high, it will check that the Rx AbortSignal goes high 2 clock cycles later. As we would like to check this at all times during simulation to correctly verify the intended behavior, it is suitable as a concurrent assertion.

```
59    sequence Rx_Abortdetect;
60      !Rx ##1 Rx [*7]; // 11111110
61    endsequence
62
63    property RX_AbortSignal;
64      @(posedge Clk) disable iff(!Rst) Rx_Abortdetect ##0 Rx_ValidFrame |=> ##2 Rx_AbortSignal;
65    endproperty
```

Figure 5: RX_AbortSignal property (Concurrent Assertion)

3 Part B

3.1 Overview of Specifications

Specification 1	Immediate Assertion
Specification 2	Immediate Assertion
Specification 3	Immediate Assertion
Specification 4	Immediate Assertion
Specification 5	Immediate Assertion
Specification 6	Immediate Assertion / Attempted Concurrent Assertion
Specification 7	Immediate Assertion / Attempted Concurrent Assertion
Specification 8	Immediate Assertion
Specification 9	Concurrent Assertion
Specification 10	Concurrent Assertion
Specification 11	Immediate Assertion
Specification 12	Concurrent Assertion
Specification 13	Concurrent Assertion
Specification 14	Immediate Assertion
Specification 15	Concurrent Assertion
Specification 16	Concurrent Assertion
Specification 17	Concurrent Assertion
Specification 18	Immediate Assertion

Table 1: Assertion type chosen for each specification

3.2 Rx Specifications

This section covers the assertions made for the receive part of the HDLC controller.

3.2.1 Specification 1-3

For the first specifications, we need to verify that the correct data is within the Rx buffer and the Rx status/control (Rx_SC) register.

Following the protocol we would expect that the data in the Rx buffer corresponds to the Rx input. This holds true unless there is a dropped frame, a frame error, or if we abort the signal through an abort-flag sequence (01111111). In these cases, we expect the Rx buffer to be zero when attempting to read it.

The individual bits for the Rx_SC register can be viewed in table 2, where bits [7] and [6] are left out as they are not in use. It is expected that the Rx_Overflow bit is high when receiving more than 128 bytes, where two of the bytes represent the Frame Check Sequence (FCS) bytes. We also expect the Rx_AbortSignal to be high when inputting an abort-flag sequence, and the Rx_FrameError bit to

be asserted when there is an error detected during the CRC calculation. When writing '1' into Rx_Drop, we expect that the whole frame is discarded. Lastly, Rx_Ready indicates that the buffer is ready to be read.

Bit:	**[5]**	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**
Field:	Rx_FCSen	Rx_Overflow	Rx_AbortSignal	Rx_FrameError	Rx_Drop	Rx_Ready
R/W:	WO	RO	RO	RO	WO	RO

Table 2: Rx Status/Control Register

Table 3 illustrates the expected bits in the RX_SC register for each of the different receive scenarios. The parentheses around bit [1] during a dropped frame is meant to indicate that the bit is '1' as we're manually writing to it, but it is expected to go down to '0' in the task where we're reading the value in the register. These expected values are used for the upcoming assertions.

Bit:	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**
Expected Values					
Normal:	0	0	0	0	1
Abort:	0	1	0	0	0
Overflow:	1	0	0	0	1
FrameError:	0	0	1	0	0
Drop:	0	0	0	(1)	0

Table 3: Expected values in Rx Status/Control Register for each scenario

Correct data in the RX databuffer and RX_SC register was verified under different conditions using immediate assertions through tasks *VerifyNormalReceive*, *VerifyAbortReceive*, *VerifyOverflowReceive*, *VerifyFrameErrorReceive*, and *VerifyDropReceive*. The tasks are called depending on the input stimulus generated in the *Receive* task - For example, under the preconditions for a which a normal receive happens, we check the registers for the correct values according to the specification through the task *VerifyNormalReceive*. Immediate assertions are chosen here as we would like to check the registers for the correct values immediately after writing to the registers. The input stimulus made in the *Receive* generating these preconditions can be found in appendix A.

VerifyNormalReceive
See Part A in section 2.

VerifyAbortReceive
See Part A in section 2.

VerifyOverflowReceive
See Part A in section 2.

VerifyFrameErrorReceive
A frame error was simulated by not adding the precomputed FCS bytes to the last 2 bytes of the input stimulus.

The expected value in the Rx_SC register in case of a frame error, is that the Rx_FrameError bit is asserted. The Rx buffer is expected to be zero.

```
task VerifyFrameErrorReceive(logic [127:0][7:0] data, int Size);
  logic [7:0] ReadData;
  automatic int RXSC_error = 0;
  automatic int RXbuff_error = 0;

  // VERIFY DATA IN RXSC REGISTER
  ReadAddress(RxSC, ReadData);
  assert (!ReadData [0]) else begin $display("VerifyFrameErrorReceive FAIL : Rx_Ready bit is not low (Immediate assertion)"); RXSC_error++; end
  assert (!ReadData [1]) else begin $display("VerifyFrameErrorReceive FAIL : Rx_Drop bit is not low (Immediate assertion)"); RXSC_error++; end
  assert (ReadData [2]) else begin $display("VerifyFrameErrorReceive FAIL : Rx_FrameError bit is not high (Immediate assertion)"); RXSC_error++; end
  assert (!ReadData [3]) else begin $display("VerifyFrameErrorReceive FAIL : Rx_AbortSignal bit is not low (Immediate assertion)"); RXSC_error++; end
  assert (!ReadData [4]) else begin $display("VerifyFrameErrorReceive FAIL : Rx_Overflow bit is not low (Immediate assertion)"); RXSC_error++; end

  // VERIFY DATA IN RX BUFFER
  for (int i = 0; i < Size; i ++) begin
    ReadAddress(RxBuff, ReadData);
    assert (ReadData == 0) else begin $display("VerifyFrameErrorReceive FAIL : Rx data buffer is not zero (Immediate assertion)"); RXbuff_error++; end
  end

  if (!RXSC_error && !RXbuff_error)
    $display("VerifyFrameErrorReceive PASS : Rx databuffer and Rx_SC contains the correct values (Immediate assertion)");
  else
    TbErrorCnt += RXSC_error + RXbuff_error;
endtask
```

Figure 6: VerifyFrameErrorReceive task (Immediate Assertion)

VerifyDropReceive A framedrop was simulated by asserting the Rx_Drop bit in the Rx_SC register. Similar to the other scenarios, it was verified using immediate assertions by calling the task after writing to the Rx_SC register. During a dropped frame, the data in the buffer should be discarded and we should read all zeros.

```
task VerifyDropReceive(logic [127:0][7:0] data, int Size);
  logic [7:0] ReadData;
  automatic int RXSC_error = 0;
  automatic int RXbuff_error = 0;

  // VERIFY DATA IN RXSC REGISTER
  ReadAddress(RxSC, ReadData);
  assert (!ReadData[0]) else begin $display("VerifyDropReceive       FAIL : Rx_Ready bit is not low (Immediate assertion)"); RXSC_error++; end
  assert (!ReadData[1]) else begin $display("VerifyDropReceive       FAIL : Rx_Drop bit is not low (Immediate assertion)"); RXSC_error++; end
  assert (!ReadData[2]) else begin $display("VerifyDropReceive       FAIL : Rx_FrameError bit is not low (Immediate assertion)"); RXSC_error++; end
  assert (!ReadData[3]) else begin $display("VerifyDropReceive       FAIL : Rx_AbortSignal bit is not low (Immediate assertion)"); RXSC_error++; end
  assert (!ReadData[4]) else begin $display("VerifyDropReceive       FAIL : Rx_Overflow bit is not low(Immediate assertion)"); RXSC_error++; end

  // VERIFY DATA IN RX BUFFER
  for (int i = 0; i < Size; i++) begin
    ReadAddress(RxBuff, ReadData);
    assert (ReadData == 0) else begin $display("VerifyDropReceive       FAIL : Rx data buffer is not zero (Immediate assertion)"); RXbuff_error++; end
  end

  if (!RXbuff_error && !RXSC_error)
    $display("VerifyDropReceive       PASS : Rx databuffer and Rx_SC contain the correct values (Immediate assertion)");
  else
    TbErrorCnt += (RXSC_error + RXbuff_error);
endtask
```

Figure 7: VerifyDropReceive task (Immediate Assertion)

3.2.2 Specification 10

See Part A in section 2.

3.2.3 Specification 12

- When a whole RX frame has been received, check if end of frame is generated.

We expect a whole frame to be received when Rx_ValidFrame goes from '1' to '0' (indicating that all bits in the message has been received), in which case the signal Rx_EoF should be asserted at the next clock. This holds true according to the logic in the RxController module. A concurrent assertion is used here as we want to continously monitor Rx_EoF. If we were to use an immediate assertion, the assertion result would not indicate anything if the assertion were to fail at other points during the simulation.

```
110   property EndOfFrame;
111     @(posedge Clk) disable iff(!Rst)  $fell(Rx_ValidFrame) |=> $rose(Rx_EoF);
112   endproperty
```

Figure 8: EndOfFrame property (Concurrent Assertion)

3.2.4 Specification 13

- When receiving more than 128 bytes, Rx Overflow should be asserted.

This was partly verified through specification 1-3 when checking the Rx_SC register through immediate assertions, but we have also checked it using a separate concurrent assertion. If there is a valid frame and a 129th Rx_NewByte, we expected Rx_Overflow to be asserted.

```
130   sequence Rx_NewByteSeq;
131     ($rose(Rx_NewByte))[->129];
132   endsequence
133
134   property Rxbuffer_Overflow;
135     @(posedge Clk) disable iff (!Rst || !Rx_ValidFrame) $rose(Rx_ValidFrame) ##0 Rx_NewByteSeq |=> $rose(Rx_Overflow);
136   endproperty
```

Figure 9: Rxbuffer_Overflow property (Concurrent Assertion)

3.2.5 Specification 14

- Rx FrameSize should equal the number of bytes received in a frame (max. 126 bytes =128 bytes in buffer – 2 FCS bytes).

An assertion was used to verify that Rx_FrameSize holds the number of bytes received in a frame. An immediate assertion was used as we're interested in

verifying this value after writing to the buffer. This was done through comparison of the data stored in the Rx_len register as this represents the size of the received frame. Alternatively one could also compare the Size integer passed in through the *Receive* task.

```
379  task VerifyFrameSize();
380    logic [7:0] ReadData;
381
382    ReadAddress(RxLen, ReadData);
383    assert (ReadData == uin_hdlc.Rx_FrameSize) $display("VerifyFrameSize
384    PASS : Rx_FrameSize equals the number of bytes received in a frame (Immediate assertion)");
385    else begin $display("VerifyFrameSize
386    FAIL : Rx_FrameSize does not equal the number of bytes received in a frame (Immediate assertion)"); TbErrorCnt++; end;
387  endtask
```

Figure 10: VerifyFrameSize task (Immediate Assertion)

3.2.6 Specification 15

- Rx Ready should indicate byte(s) in RX buffer is ready to be read.

We expect Rx_Ready to be asserted if Rx_EoF was high at the previous clock cycle indicating that a whole frame had been received. This needs to hold true at all times during simulation to detect a wrongly asserted Rx_Ready, so a concurrent assertion was chosen.

```
142    property ReadyCheck;
143      @(posedge Clk) disable iff(!Rst) $rose(Rx_Ready) |=>$past(Rx_EoF) ;
144    endproperty
```

Figure 11: ReafdyChek property (Concurrent Assertion)

3.2.7 Specification 16

- Non-byte aligned data or error in FCS checking should result in frame error.

For the correlation between a FCS error and Rx_FrameError being asserted, a concurrent assertion was used to check this at all times during simulation. We unfortunately did not have time to create stimulus for a case where there is non-byte aligned data, so we were not able to verify the other part.

```
171    property RX_FrameError;
172      @(posedge Clk) disable iff (!Rst) Rx_FCSerr && Rx_FCSen |=> $rose(Rx_FrameError);
173    endproperty
```

Figure 12: RX_FrameError property (Concurrent Assertion)

3.3 Tx Specifications

This section covers the assertions made for the transmit part of the HDLC controller. Both immediate and concurrent assertions have been used.

The Tx_SC register can be viewed in table 4, where the upper bits are not in use similar to the Rx_SC register. In particular, Tx_AbortFrame and Tx_Enable has been used to simulate abortion and transmission of frame in the *Transmit* task used for creating stimulus. Tasks are then called upon to verify the correct output, using immediate assertions.

Bit:	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**
Field:	Tx_Full	Tx_AbortedTrans	Tx_AbortFrame	Tx_Enable	Tx_Done
R/W:	RO	RO	WO	WO	RO

Table 4: Tx Status/Control Register

3.3.1 Specification 4

- Correct TX output according to written TX buffer.

Preconditions When verifying that the Tx output is correct according to the written Tx buffer, one has to take into consideration the added zeros whenever five consecutive ones are transmitted. For example, if the input stream to the buffer is *01111101*, the expected output will be *011111001*. Therefore, in both upcoming tasks, we are passing in a zeroinserted data array when comparing the actual output instead of directly comparing the written Tx data. This replication of the zero insertion can be viewed in appendix A.1. Other preconditions that are made before verifying the buffers, are that Tx_Enable bit is asserted to allow for transmission, and that stimulus for the start-flag is generated.

VerifyNormalTransmit
In the case of a normal transmit, the Tx output should correspond to the written buffer, but with zeros inserted. For this assertion, we check bit by bit if the values precomputed zeroinserted values correspond with the actual Tx output bit. Unlike earlier we cannot compare the values byte-by-byte anymore as they won't align because of the zero instertion. Additional bits added as a result of zero insertions are also precomputed to ensure that we also loop over the additional bits. The computation of additional bits can be viewed in appendix B.

```
178  task VerifyNormalTransmit(logic [149:0][7:0] data, int Size, logic [3:0] extra_bits, logic [7:0] extra_bytes);
179    logic [7:0] ReadData;
180    automatic int Txbuff_error = 0;
181    automatic int newSize = Size + extra_bytes;
182    automatic int ending_bit;
183
184    for (int i = 0; i < newSize; i++) begin
185      if (i == (newSize - 1) && (extra_bits != 0))
186        ending_bit = extra_bits;
187      else
188        ending_bit = 8;
189
190      for (int j = 0; j < ending_bit; j++) begin
191        ReadData[j] = uin_hdlc.Tx;
192        assert(data[i][j] == uin_hdlc.Tx) else begin $display("VerifyNormalTransmit
193        FAIL : TX output does not match for written bit[%0d][%0d] (Immediate assertion)", i, j); Txbuff_error++; end;
194        @(posedge uin_hdlc.Clk);
195      end
196    end
197
198    if (!Txbuff_error)
199      $display("VerifyNormalTransmit    PASS : TX output is correct according to written TX buffer (Immediate assertion)");
200    else
201      TbErrorCnt += Txbuff_error;
202  endtask
```

Figure 13: VerifyNormalTransmit task (Immediate assertion)

VerifyAbortTransmit

An additional precondition when verifying an aborted transmission is that the Tx_AbortFrame bit in the Tx_SC register is asserted in the stimulus task, this will generate the abort-flag sequence.

Our assertions checks that the Tx output is '1' after an aborted transmission. Note that this task is not called until after the abort-flag sequence has been transmitted, otherwise we would expect the assertion to fail as the Tx output will be *01111111*.

```
203  task VerifyAbortTransmit(int Size);
204  logic [7:0] ReadData;
205  automatic int Txbuff_error = 0;
206
207  for (int i = 0; i < Size; i++) begin
208    for (int j = 0; j < 8; j++) begin
209      @(posedge uin_hdlc.Clk);
210      ReadData[j] = uin_hdlc.Tx;
211    end
212    assert(ReadData == 8'b11111111) else begin $display("VerifyAbortTransmit
213    FAIL : TX output does not match for written byte[%0d] (Immediate assertion)", i); Txbuff_error++; end;
214  end
215
216  if (!Txbuff_error)
217    $display("VerifyAbortTransmit     PASS : TX output is correct according to written TX buffer (Immediate assertion)");
218  else
219    TbErrorCnt += Txbuff_error;
220  endtask
```

Figure 14: VerifyAbortTransmit task (Immediate assertion)

3.3.2 Specification 5

- Start and end of frame pattern generation (Start and end flag: 0111 1110).

Start and end frame pattern generation is checked immediately after writing to the buffer and setting the Tx_Enable bit high. We then compare the output to the flagsequence *01111110*. Seeing as the specification is about verifying correct behavior at only two points during simulation (start and end frame), we thought that immediate assertions were suitable.

```
227   task VerifyStartEndFlag(int start);
228     logic [7:0] flagsequence;
229     logic [7:0] byte_received;
230     flagsequence = 8'b01111110;
231
232     if (start)
233       wait(!uin_hdlc.Tx && uin_hdlc.Tx_ValidFrame);
234     else begin
235       wait(!uin_hdlc.Tx_ValidFrame);
236       @(posedge(uin_hdlc.Clk));
237     end
238
239     for (int i = 0; i < 8; i++) begin
240       byte_received[i] = uin_hdlc.Tx;
241       @(posedge uin_hdlc.Clk);
242     end
243
244     if (start) begin
245       assert(byte_received == flagsequence) $display("VerifyStartFlag
246       PASS : Flagsequence was generated at the start of frame (Immediate assertion)");
247       else begin $display("VerifyStartFlag
248       FAIL : Flagsequence was not generated at the start of frame (Immediate assertion)");
249       TbErrorCnt++; end;
250     end
251     else begin
252       assert(byte_received == flagsequence) $display("VerifyEndFlag
253       PASS : Flagsequence was generated at the end of frame (Immediate assertion)");
254       else begin $display("VerifyEndFlag
255       FAIL : Flagsequence was not generated at the end of frame (Immediate assertion)");
256       TbErrorCnt++; end;
257     end
258   endtask
```

Figure 15: VerifyStartEndFlag task (Immediate Assertion)

3.3.3 Specification 6

- Zero insertion and removal for transparent transmission.

Verifying that the zeroinsertion works is partly done in specification 4 when we monitor the Tx output, but a separate task has been made for this that continously checks if the output is '0' after five consecutive '1' have been detected.

```
task VerifyZeroInsertion(input logic [127:0][7:0] datain, int Size);
  logic [4:0] PrevData;
  automatic int insertion_error = 0;
  PrevData = '0;

  for (int i = 0; i < Size; i++) begin
    for (int j = 0; j < 8; j++) begin
      PrevData[4] = datain[i][j];
      if(&PrevData) begin
        @(posedge uin_hdlc.Clk);
        assert (uin_hdlc.Tx == 1'b0)
        else begin $display("VerifyZeroInsertion
        FAIL : TX output bit was not zero when detecting five 1's (Immediate assertion)"); insertion_error++; end;
        PrevData = PrevData >> 1;
        PrevData[4] = 1'b0;
      end

      @(posedge uin_hdlc.Clk);
      PrevData = PrevData >> 1;
    end
  end

  if (!insertion_error)
    $display("VerifyZeroInsertion
    PASS : TX output bit was zero when detecting five 1's (Immediate assertion)");
  else
    TbErrorCnt += insertion_error;
endtask
```

Figure 16: VerifyZeroInsertion task (Immediate Assertion)

Typically a concurrent assertion checking for a '0' Tx output whenever a sequence of five '1's are detected would be suitable for this. However we think that in this case it was doable to also use an immediate assertion. An attempt of a concurrent assertion is discussed later.

3.3.4 Specification 7

- Idle pattern generation and checking (1111 1111 when not operating).

Verifying that the idle pattern is generated when the controller is unoperating was checked using an immediate assertion within a task that is called before writing any data to the Tx buffer. For extra coverage it could have be called after all data had been transmitted and present at the output, which is a case where we also expect an idle transmission.

```
task VerifyIdleSequence();
  logic [7:0] idlesequence;
  logic [7:0] byte_received;
  idlesequence = 8'b11111111;

  for (int i = 0; i < 8; i++) begin
    @(posedge uin_hdlc.Clk);
    byte_received[i] = uin_hdlc.Tx;
  end

  assert(byte_received == idlesequence) $display("VerifyIdleSequence      PASS : Idlesequence was generated when the controller is unoperating");
                                        else begin $display("VerifyIdleSequence      FAIL : Idlesequence was not generated when the controller is unoperating");
                                        TbErrorCnt++; end;
endtask
```

Figure 17: VerifyIdleSequence task(Immediate Assertion)

Alternatively this would make more sense as a concurrent assertion as we would like to check this continuously and not only during the function call in our testbench. Our current assertion verifies that the idle pattern is generated prior to enabling transmission, but we do not check it at all times during simulation. This is a flaw, and is discussed more in detail later in the report.

3.3.5 Specification 8

- Abort pattern generation and checking (1111 1110). Remember that the 0 must be sent first.

The abort-flag sequence was generated by asserting the Tx_AbortFrame bit in the TX_SC register, before then immediately verifying that the abort sequence is generated through the *VerifyAbortTransmit* task.

```
320  task VerifyAbortSequence();
321    logic [7:0] abortsequence;
322    logic [7:0] byte_received;
323    logic [7:0] reading;
324    abortsequence = 8'b11111110;
325
326    wait(uin_hdlc.Tx_AbortedTrans && uin_hdlc.Tx_FCSDone && !uin_hdlc.Tx_DataAvail);
327    for (int i = 0; i < 8; i++) begin
328      @(posedge uin_hdlc.Clk);
329      byte_received[i] = uin_hdlc.Tx;
330    end
331
332    assert(byte_received == abortsequence) $display("VerifyAbortSequence     PASS : Abortsequence was generated when aborting transmission");
333                                           else begin $display("VerifyAbortSequence     FAIL : Abortsequence was not generated when aborting transmission");
334                                           TbErrorCnt++; end;
335  endtask
```

Figure 18: VerifyAbortSequence task (Immediate Assertion)

Because this is something that would need to be checked only once during every iteration of our stimulus task (*Transmit*), we thought it was justifiable to use an immediate assertion. Alternatively a concurrent assertion would also be an option as we simply need to check that the abort-flag sequence is detected whenever Tx_AbortedTrans is asserted.

```
139    sequence Tx_Abortdetect;
140      !Tx ##1 Tx [*7];
141    endsequence
142
143    property Tx_AbortSequence;
144      @(posedge Clk) disable iff(!Rst) $rose(Tx_AbortedTrans) |-> ##2 Tx_Abortdetect;
145    endproperty
```

Figure 19: Tx_AbortSequence property (Concurrent Assertion)

3.3.6 Specification 9

- When aborting frame during transmission, Tx AbortedTrans should be asserted.

For verifying that Tx_AbortedTrans is asserted after Tx_AbortFrame, it was

suitable to use a concurrent assertion to ensure that this held true at all times during simulation.

```
property TX_AbortFrame;
  @(posedge Clk) disable iff (!Rst) Tx_AbortFrame |-> ##2 $rose(Tx_AbortedTrans);
endproperty
```

Figure 20: TX_AbortFrame property (Concurrent Assertion)

3.3.7 Specification 11

- CRC generation and Checking

We we're uncertain about what was meant about the CRC checking - whether it was verifying that the checksum of message+CRC is zero, or if it was to verify that the same FCS bytes generated are also present at the output. We verified the latter, by comparing a zeroinserted version of the precomputed FCS bytes with the actual output after the whole message (excluding the FCS bytes) had been received. This was suitable as an immediate assertion, as we could verify the correct output immediately after the *VerifyNormalTransmit* had finished executing.

```
task VerifyFCSTransmit(logic [23:0] data, logic [3:0] extra_bits);
  logic [7:0] ReadData;
  automatic int Txbuff_error = 0;
  automatic int newSize = 16 + extra_bits;
  automatic int ending_bit;

  for (int i = 0; i < newSize; i++) begin
    assert(data[i] == uin_hdlc.Tx)
    else begin $display("VerifyFCSTransmit       FAIL : FCS output does not match for written bit[%0d]", i); Txbuff_error++; end;
    @(posedge uin_hdlc.Clk);
  end


  if (!Txbuff_error)
    $display("VerifyFCSTransmit       PASS : FCS output is correct according to generated FCS bytes");
  else
    TbErrorCnt += Txbuff_error;
endtask
```

Figure 21: VerifyFCSTransmit task (Immediate assertion)

3.3.8 Specification 17

- Tx Done should be asserted when the entire TX buffer has been read for transmission.

It is expected that Tx_Done should be asserted when there is no longer data available in the Tx buffer. This was done using a concurrent assertion to ensure Tx_Done was not asserted at any other points during simulation.

```
150   property TxbufferDone;
151     @(posedge Clk) disable iff (!Rst) $fell(Tx_DataAvail) |-> Tx_Done;
152   endproperty
```

Figure 22: TxbufferDone property (Concurrent Assertion)

3.3.9 Specification 18

- Tx Full should be asserted after writing 126 or more bytes to the TX buffer (overflow).

This specification was checked using an immediate assertion by verifying that Tx_Full was asserted when passing in a Size higher than 126 defined as MAX_SIZE. We only need to check this once for each transmission, so no need to use concurrent assertion to check this continously.

```
414   task VerifyTxFull(int Size);
415     logic [7:0] MAX_SIZE;
416     MAX_SIZE = 8'h7E;
417     //$display("checking if %0d > %0d...", Size, MAX_SIZE); // f
418
419     if (Size > MAX_SIZE)
420       assert(uin_hdlc.Tx_Full == 1'b1) $display ("VerifyTxFull
421       PASS : Tx_Full was asserted after writing 126 or more byte
422       else begin $display("VerifyTxFull
423       FAIL : Tx_Full was not asserted after writing 126 or more
424   endtask
```

Figure 23: VerifyTxFull task (Immediate Assertion)

4 Results

Result of simulation show zero assertion errors and a coverage of 94.31% with our given stimulus and the coverpoints used.

```
# ****************************************************************
# *                          *                                   *
# *    Assertion Errors: 0   *    Coverage: 94.31 %              *
# *                          *                                   *
# ****************************************************************
```

Figure 24: Simulation result: Assertion Error and Coverage

The main source of reduction in coverage was DataIn, DataOut and Rx_FrameSize.

```
96    Coverpoint DataIn                                87.50%
97        covered/total bins:                            112
98        missing/total bins:                             16
99        % Hit:                                       87.50%
100   Coverpoint DataOut                               86.71%
101       covered/total bins:                            111
102       missing/total bins:                             17
103       % Hit:                                       86.71%
```

```
162   Coverpoint Rx_FrameSize                          6.29%
163       covered/total bins:                              8
164       missing/total bins:                            119
165       % Hit:                                       6.29%
```

Figure 25: Coverage point: DataIn, DataOut and Rx_FrameSize.

5 Discussion

5.1 Coverage

DataIn, DataOut and Rx_FrameSize being the culprit for a lower coverage is reasonable considering there are only 14 iterations of the *Receive/Transmit* tasks in our simulation code. Having more iterations would increase the coverage as there would be a broader range of stimulus applied.

5.2 Possible improvements / Other alternatives

The result of our assertions show a working design under the given stimulus applied, but we think that some assertions could be improved. In particular we think that the idle pattern and zero insertion (specification 6 and 7) could have been better suited as concurrent assertions. This especially applies to the idle pattern, which at it's current state using immediate assertions is only being evaluated at specific time points during the simulation, when it instead should be checked continuously at all times. Initially we tried to make a concurrent assertion for the zero insertion as seen in figure 26.

```
147   sequence fiveones;
148     Tx [*5];
149   endsequence
150
151   sequence fivevalidframes;
152     Tx_ValidFrame [*5];
153   endsequence
154
155   property Tx_ZeroInsertion;
156   @(posedge Clk) disable iff(!Rst) fiveones and fivevalidframes |=> !Tx;
157   endproperty
```

Figure 26: Tx_ZeroInsertion property attempt (Concurrent Assertion)

The problem with this approach is that the assertion fails during the flag-sequence at the start of a frame, as Tx_ValidFrame is high during this phase.

6 Conclusion

Our assertion prove to be working, but certain assertions could arguably have been made as concurrent instead of immediate. Results show zero assertion fails and a coverage of 94.31%. Further improvements for coverage could be made through a more extensive testbench generating more input stimulus.

A Main functions

A.1 Receive

```
697   task Receive(int Size, int Abort, int FCSerr, int NonByteAligned, int Overflow, int Drop, int SkipRead);
698     logic [127:0][7:0] ReceiveData;
699     logic       [15:0] FCSBytes;
700     logic   [2:0][7:0] OverflowData;
701     string msg;
702
703     if(Abort)
704       msg = "- Abort";
705     else if(FCSerr)
706       msg = "- FCS error";
707     else if(NonByteAligned)
708       msg = "- Non-byte aligned";
709     else if(Overflow)
710       msg = "- Overflow";
711     else if(Drop)
712       msg = "- Drop";
713     else if(SkipRead)
714       msg = "- Skip read";
715     else
716       msg = "- Normal";
717     $display("************************************************************");
718     $display("%t - Starting task Receive with messageSize %0d %s", $time, Size, msg);
719     $display("************************************************************");
720
721     for (int i = 0; i < Size; i++) begin
722       ReceiveData[i] = $urandom;
723     end
724     ReceiveData[Size]   = '0;
725     ReceiveData[Size+1] = '0;
726
727     //Calculate FCS bits;
728     GenerateFCSBytes(ReceiveData, Size, FCSBytes);
729     if (!FCSerr) // Added condition -J
730       ReceiveData[Size]   = FCSBytes[7:0];
731       ReceiveData[Size+1] = FCSBytes[15:8];
732
733     //Enable FCS
734     if(!Overflow && !NonByteAligned)
735       WriteAddress(RxSC, 8'h20);
736     else
737       WriteAddress(RxSC, 8'h00);
```

Figure 27: Receive task (1/2)

```
739     //Generate stimulus
740     InsertFlagOrAbort(1);
741
742     MakeRxStimulus(ReceiveData, Size + 2);
743
744     if(Overflow) begin
745       OverflowData[0] = 8'h44;
746       OverflowData[1] = 8'hBB;
747       OverflowData[2] = 8'hCC;
748       MakeRxStimulus(OverflowData, 3);
749     end
750
751     if(Abort) begin
752       InsertFlagOrAbort(0);
753     end else begin
754       InsertFlagOrAbort(1);
755     end
756
757
758     @(posedge uin_hdlc.Clk);
759     uin_hdlc.Rx = 1'b1;
760
761     repeat(8)
762       @(posedge uin_hdlc.Clk);
763
764     if (!FCSerr)
765       VerifyFrameSize();
766
767     if(Abort)
768       VerifyAbortReceive(ReceiveData, Size);
769     else if(Overflow)
770       VerifyOverflowReceive(ReceiveData, Size);
771     else if (FCSerr)
772       VerifyFrameErrorReceive(ReceiveData, Size);
773     else if (Drop) begin
774       WriteAddress(RxSC, 8'h02);
775       VerifyDropReceive(ReceiveData, Size);
776     end
777     else if(!SkipRead)
778       VerifyNormalReceive(ReceiveData, Size);
779
780     #5000ns;
781   endtask
```

Figure 28: Receive task (2/2)

A.2 Transmit

```
624   task Transmit(int Size, int Abort);
625     logic [127:0][7:0] TransmitData;
626     logic [149:0][7:0] zeroinserted_data;
627     logic [23:0] zeroinserted_FCS;
628     logic [15:0] FCSBytes;
629     logic [7:0] zeros_inserted, extra_bytes;
630     logic [3:0] extra_bits;
631     string msg;
632
633     if(Abort)
634       msg = "- Abort";
635     else
636       msg = "- Normal";
637     $display("************************************************************");
638     $display("%t - Starting task Transmit with messageSize %0d %s", $time, Size, msg);
639     $display("************************************************************");
640
641     TransmitData = 'x;
642
643     for (int i = 0; i < Size; i++) begin
644         TransmitData[i] = $urandom;
645     end
646
647
648     ZeroInsertion(TransmitData, FCSBytes, 0, Size, zeroinserted_data, zeroinserted_FCS, zeros_inserted);
649     ComputeExtraBits(zeros_inserted, extra_bits, extra_bytes);
650
651     TransmitData[Size]   = '0;
652     TransmitData[Size+1] = '0;
653
654     GenerateFCSBytes(TransmitData, Size, FCSBytes);
655
656     VerifyIdleSequence();
657
658     wait(uin_hdlc.Tx_Done);
659     for (int i = 0; i < Size; i++) begin
660         WriteAddress(TxBuff, TransmitData[i]);
661     end
662
663     VerifyTxFull(Size);
664
665     WriteAddress(TxSC, 8'h02);
666
```

Figure 29: Transmit task (1/2)

```
666
667     VerifyStartEndFlag(1);
668
669     if (Abort) begin
670       WriteAddress(TxSC, 8'h04);
671       VerifyAbortSequence();
672       VerifyAbortTransmit(Size);
673     end
674     else begin
675       fork
676         begin
677           VerifyZeroInsertion(TransmitData, Size);
678         end
679         begin
680           VerifyNormalTransmit(zeroinserted_data, Size, extra_bits, extra_bytes);
681           ZeroInsertion(TransmitData, FCSBytes, 1, Size, zeroinserted_data, zeroinserted_FCS, zeros_inserted);
682           ComputeExtraBits(zeros_inserted, extra_bits, extra_bytes);
683           VerifyFCSTransmit(zeroinserted_FCS, extra_bits);
684         end
685         begin
686           VerifyStartEndFlag(0);
687         end
688       join
689     end
690
691     repeat(8)
692       @(posedge uin_hdlc.Clk);
693
694   endtask
```

Figure 30: Transmit task (2/2)

A.3 Simulation code

```
427     $display("================================================================");
428     $display("%t - Starting Test Program", $time);
429     $display("================================================================");
430
431     Init();
432
433     //Receive: Size, Abort, FCSerr, NonByteAligned, Overflow, Drop, SkipRead
434     Receive($urandom_range(126, 0), 0, 0, 0, 0, 0, 0); //Normal
435     Receive($urandom_range(126, 0), 1, 0, 0, 0, 0, 0); //Abort
436     Receive(126, 0, 0, 0, 1, 0, 0); //Overflow
437     Receive($urandom_range(126, 0), 0, 0, 0, 0, 0, 0); //Normal
438     Receive($urandom_range(126, 0), 1, 0, 0, 0, 0, 0); //Abort
439     Receive(126, 0, 0, 0, 1, 0, 0); //Overflow
440     Receive($urandom_range(126, 0), 0, 0, 0, 0, 0, 0); //Normal
441     Receive($urandom_range(126, 0), 0, 1, 0, 0, 0, 0); //FCSError
442     Receive($urandom_range(126, 0), 0, 0, 0, 0, 1, 0); //Drop
443
444
445
446     Transmit($urandom_range(126, 0), 0); // Normal
447     Transmit($urandom_range(126, 0), 0); // Normal
448     Transmit($urandom_range(126, 0), 0); // Normal
449     Transmit(126, 0); // Normal
450     Transmit($urandom_range(126, 0), 1); // Abort
451
452     $display("================================================================");
453     $display("%t - Finishing Test Program", $time);
454     $display("================================================================");
455     $stop;
456   end
457
458   final begin
459
460     $display("================================================================*");
461     $display("*                                                               *");
462     $display("* \tAssertion Errors: %0d\t  * \tCoverage: %0.2f %%      *", TbErrorCnt + uin_hdlc.ErrCntAssertions, gr22_cg.get_inst_coverage());
463     $display("*                                                               *");
464     $display("================================================================*");
465
466   end
```

Figure 31: Simulation code

B Supplementing code

B.1 Replicating Zero-Insertion

B.1.1 Inserting zeros

```
561  task ZeroInsertion(input logic [127:0][7:0] datain, input logic [15:0] FCSBytes, int FCSorMessage, int Size,
562  output logic[149:0][7:0] zeroinserted_data, output logic [23:0] zeroinserted_FCS, output logic [7:0] zeros_inserted);
563    static int count;
564    automatic int old_i = 0;
565    automatic int old_j = 0;
566    automatic int FCS_index = 0;
567    automatic int extra_Size = 0;
568    zeros_inserted = 0;
569    zeroinserted_data = 0;
570
571    if (!FCSorMessage) begin
572      count = 0;
573      for (int i = 0; i < 150; i++) begin
574        for (int j = 0; j < 8; j++) begin
575          if (count == 5) begin
576            count = 0;
577            zeroinserted_data[i][j] = 1'b0;
578            zeros_inserted++;
579          end
580          else if (datain[old_i][old_j] == 1'b1) begin
581            zeroinserted_data[i][j] = datain[old_i][old_j];
582            if (old_j < 7)
583              old_j++;
584            else begin
585              old_j = 0;
586              old_i++;
587            end
588            count++;
589          end
590          else begin
591            zeroinserted_data[i][j] = datain[old_i][old_j];
592            count = 0;
593            if (old_j < 7)
594              old_j++;
595            else begin
596              old_j = 0;
597              old_i++;
598            end
599          end
600        end
601      end
602    end
```

Figure 32: ZeroInsertion task (1/2)

```
603     else begin
604       old_i = 0;
605       zeros_inserted = 0;
606       for (int i = 0; i < 24; i++) begin
607         if (count == 5) begin
608           count = 0;
609           zeroinserted_FCS[i] = 1'b0;
610           zeros_inserted++;
611         end
612         else if (FCSBytes[i] == 1'b1) begin
613           zeroinserted_FCS[i] = FCSBytes[old_i];
614           count++;
615           old_i++;
616         end
617         else begin
618           zeroinserted_FCS[i] = FCSBytes[old_i];
619           count = 0;
620           old_i++;
621         end
622       end
623     end
624   endtask
```

Figure 33: ZeroInsertion task (2/2)

B.1.2 Computing additional bits in output

```
551   task ComputeExtraBits(logic [7:0] zeros_inserted, output logic [3:0] extra_bits, output logic [7:0] extra_bytes);
552     automatic int exceeding_byte;
553     automatic int totalbits;
554     extra_bits = zeros_inserted % 8;
555     extra_bytes = zeros_inserted / 8;
556
557     if (extra_bits != 0)
558       extra_bytes += 1;
559   endtask
```

Figure 34: ComputeExtraBits task

C Covergroup

```
804  covergroup hdlcproject_cg() @(posedge uin_hdlc.Clk);
805    Rx_AbortSignal : coverpoint uin_hdlc.Rx_AbortSignal {
806      bins No_RxAbort = {0};
807      bins RxAbort = {1};
808    }
809    Rx_FCSen : coverpoint uin_hdlc.Rx_FCSen {
810      bins No_RxFCSen = {0};
811      bins RxFCSen = {1};
812    }
813    Rx_FCSerr : coverpoint uin_hdlc.Rx_FCSerr {
814      bins No_RxFCSerr = {0};
815      bins RxFCSerr = {1};
816    }
817    Rx_FrameError : coverpoint uin_hdlc.Rx_FrameError {
818      bins No_RxFrameError = {0};
819      bins RxFrameError = {1};
820    }
821    Rx_Overflow : coverpoint uin_hdlc.Rx_Overflow {
822      bins No_RxOverflow = {0};
823      bins RxOverflow = {1};
824    }
825    Rx_Ready : coverpoint uin_hdlc.Rx_Ready {
826      bins No_RxReady = {0};
827      bins RxReady = {1};
828    }
829    Rx_Drop : coverpoint uin_hdlc.Rx_Drop {
830      bins No_RxDrop = {0};
831      bins RxDrop = {1};
832    }
833    Rx_ValidFrame : coverpoint uin_hdlc.Rx_ValidFrame {
834      bins No_RxValidFrame = {0};
835      bins RxValidFrame = {1};
836    }
837    Rx_EoF : coverpoint uin_hdlc.Rx_EoF {
838      bins No_RxEoF = {0};
839      bins RxEoF = {1};
840    }
841    Rx_FrameSize : coverpoint uin_hdlc.Rx_FrameSize {
842      bins RxFrameSize[] = {[0:126]};
843    }
```

Figure 35: Coverage group, part 1

```
844     Rx_FlagDetect : coverpoint uin_hdlc.Rx_FlagDetect {
845       bins No_RxFlagDetect = {0};
846       bins RxFlagDetect = {1};
847     }
848     Tx_Full : coverpoint uin_hdlc.Tx_Full {
849       bins No_TxFull = {0};
850       bins TxFull = {1};
851     }
852     Tx_AbortedTrans : coverpoint uin_hdlc.Tx_AbortedTrans {
853       bins No_TxAbortedTrans = {0};
854       bins TxAbortedTrans = {1};
855     }
856     Tx_AbortFrame : coverpoint uin_hdlc.Tx_AbortFrame {
857       bins No_TxAbortFrame = {0};
858       bins TxAbortFrame = {1};
859     }
860     Tx_Done : coverpoint uin_hdlc.Tx_Done {
861       bins No_TxDone = {0};
862       bins TxDone = {1};
863     }
864     Tx_Enable : coverpoint uin_hdlc.Tx_Enable {
865       bins No_TxEnable = {0};
866       bins TxEnable = {1};
867     }
868     Tx_DataAvail : coverpoint uin_hdlc.Tx_DataAvail {
869       bins No_TxDataAvail = {0};
870       bins TxDataAvail = {1};
871     }
872     Tx_ValidFrame : coverpoint uin_hdlc.Tx_ValidFrame {
873       bins No_TxValidFrame = {0};
874       bins TxValidFrame = {1};
875     }
876     Tx_FCSDone : coverpoint uin_hdlc.Tx_FCSDone {
877       bins No_TxFCSDone = {0};
878       bins TxFCSDone = {1};
879     }
880     DataIn : coverpoint uin_hdlc.DataIn {
881       bins datain[] = {[0:127]};
882     }
883     DataOut : coverpoint uin_hdlc.DataOut {
884       bins dataout[] = {[0:127]};
885     }
886   endgroup
```

Figure 36: Coverage group, part 2